

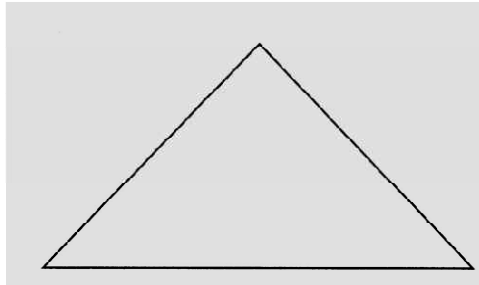
# **CSE 2017 Data Structures and Lab**

## **Lecture #10: Heap**

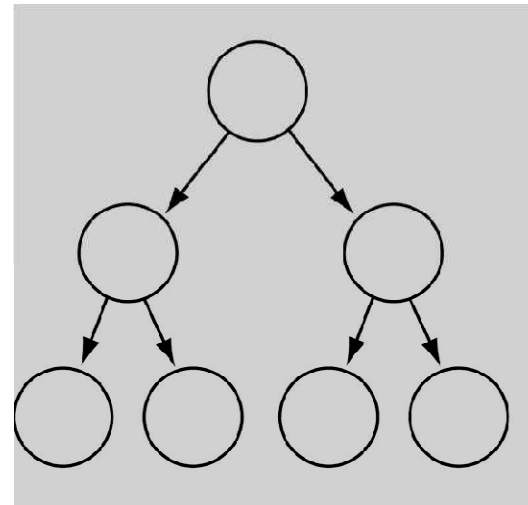
Eun Man Choi

# Full Binary Tree

- Every non-leaf node has two children
- Leaves are on the same level

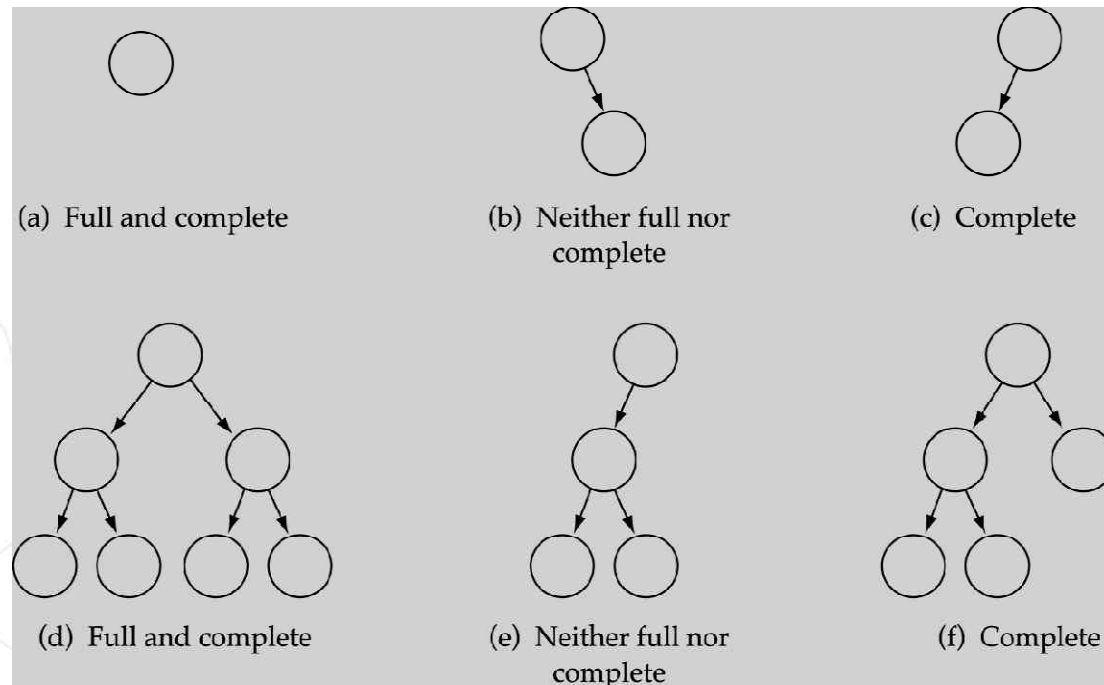
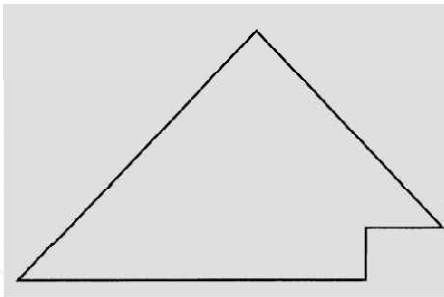


**Full Binary Tree**



# Complete Binary Tree

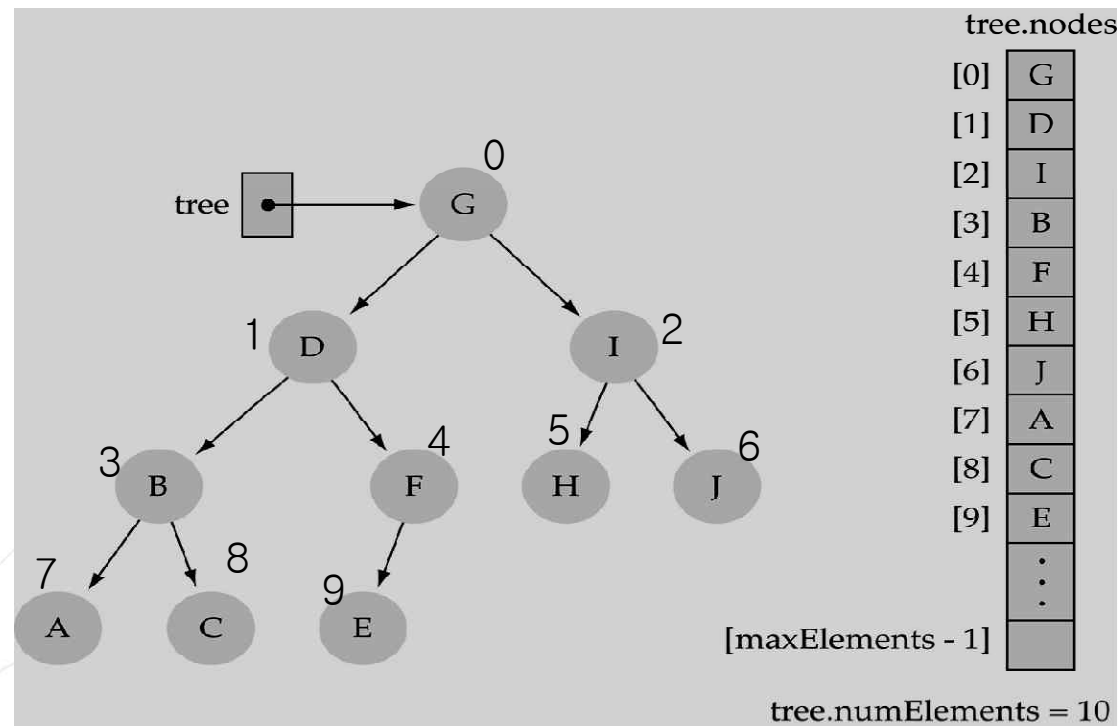
- (1) A binary tree that is either full or full through the next-to-last level
- (2) The last level is full **from left to right** (i.e., leaves are as far to the left as possible)



# Array-based representation of binary trees

- Memory savings (i.e., no pointers)
- Preserve parent-child relationships

Store: (i) level by level, and (ii) left to right



# Array-based representation of binary trees (cont.)

- Parent-child relationships:

- left child of  $tree.nodes[index] = tree.nodes[2*index+1]$
- right child of  $tree.nodes[index] = tree.nodes[2*index+2]$
- parent node of  $tree.nodes[index] = tree.nodes[(index-1)/2]$

(int division)

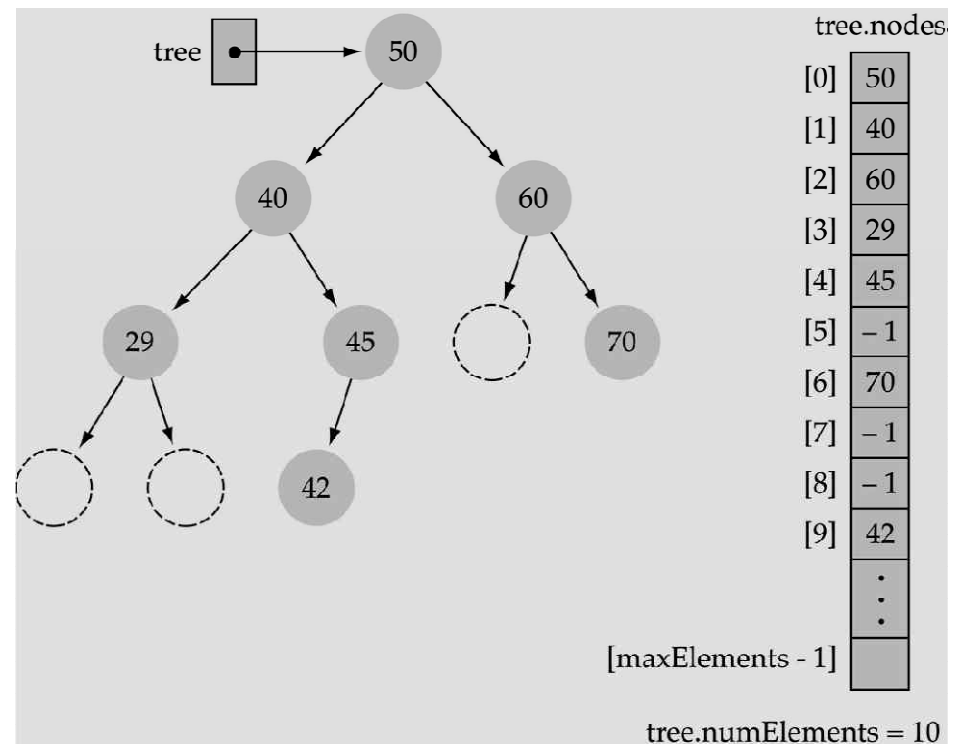
- Leaf nodes:

- $tree.nodes[numElements/2]$  to  $tree.nodes[numElements - 1]$

# Array-based representation of binary trees (cont.)

- Full or complete trees can be implemented efficiently using an array-based representation (i.e., elements occupy contiguous array slots).

**"Dummy nodes"** are required for trees which are not full or complete.

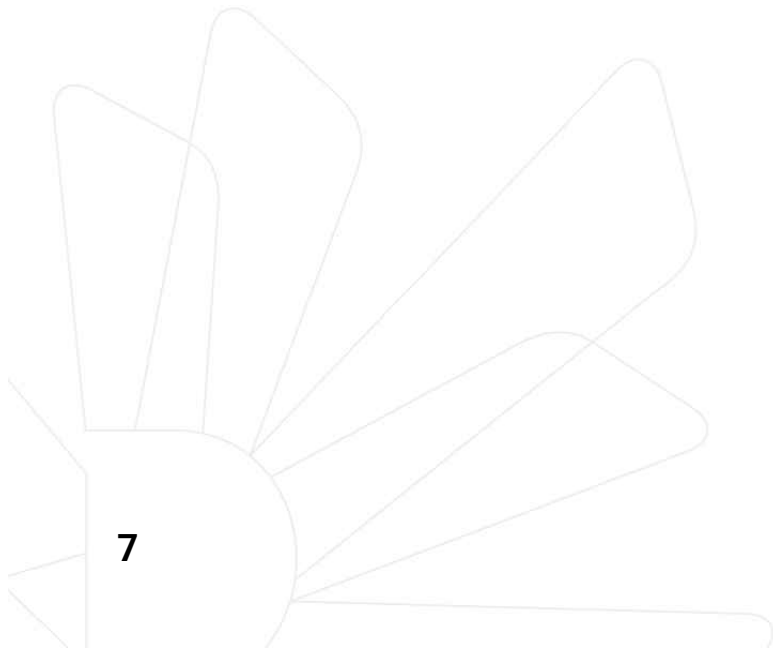


# What is a heap?

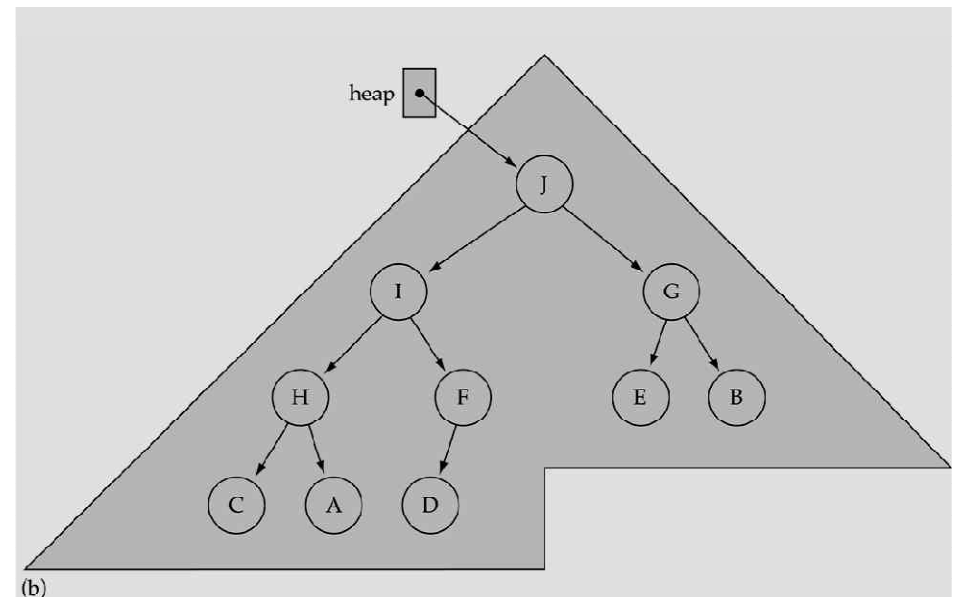
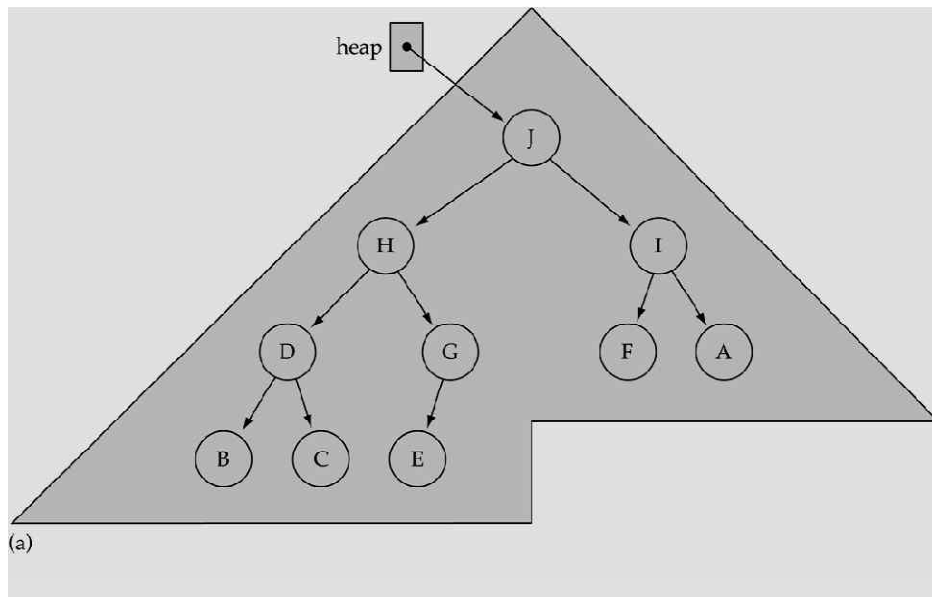
- It is a binary tree with the following properties:

Property 1: it is a complete binary tree

Property 2: (heap property): the value stored at a node is greater or equal to the values stored at the children



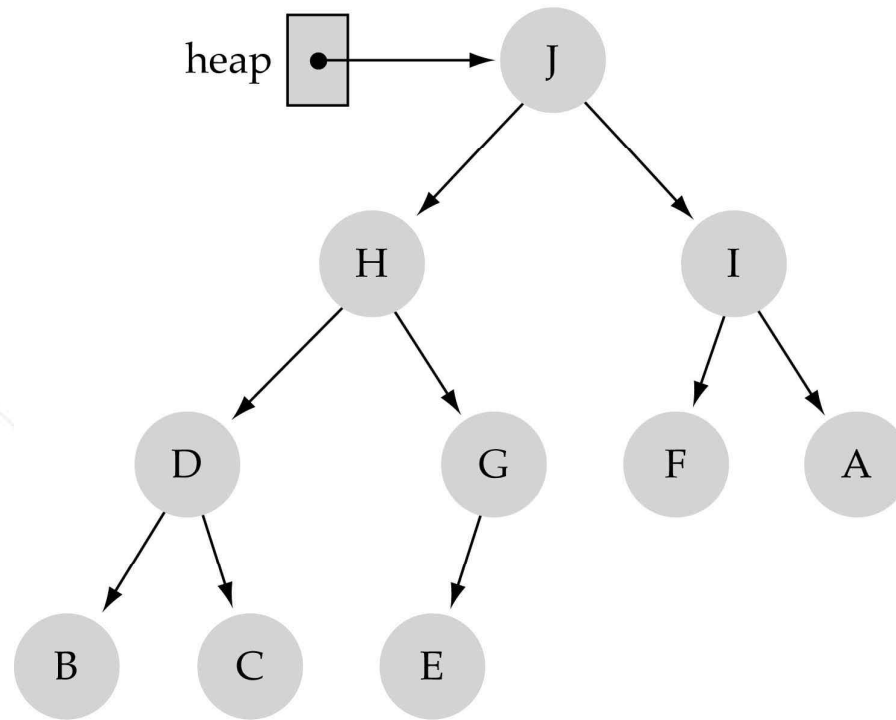
# Not **unique!**





# Largest heap element

- From *Property 2*, the largest value of the heap is always stored at the root

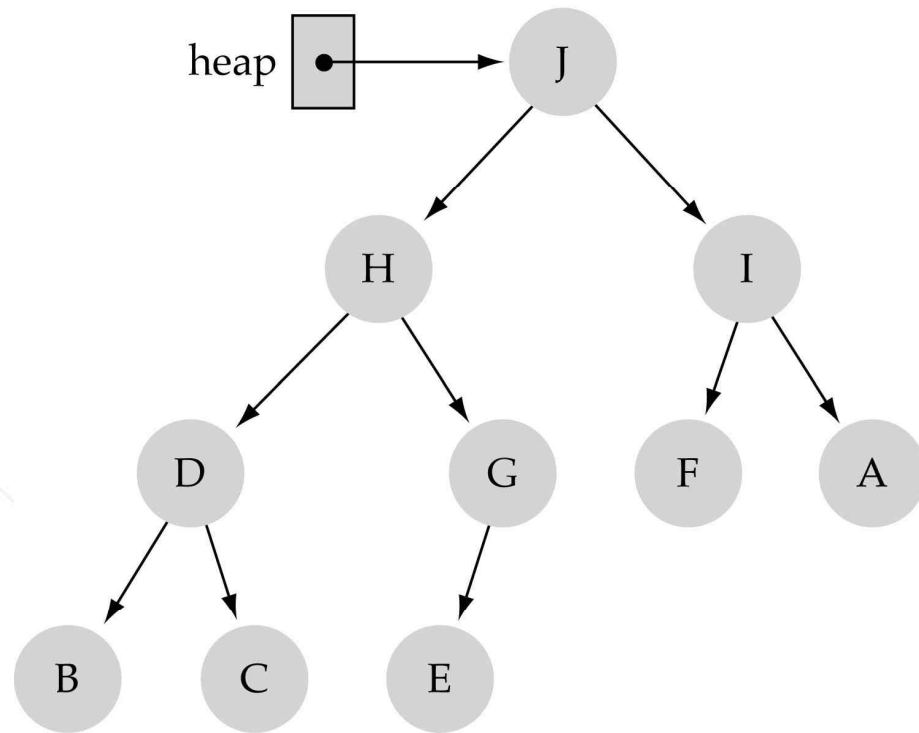


heap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

# Heap implementation

- Heaps are always implemented as arrays!



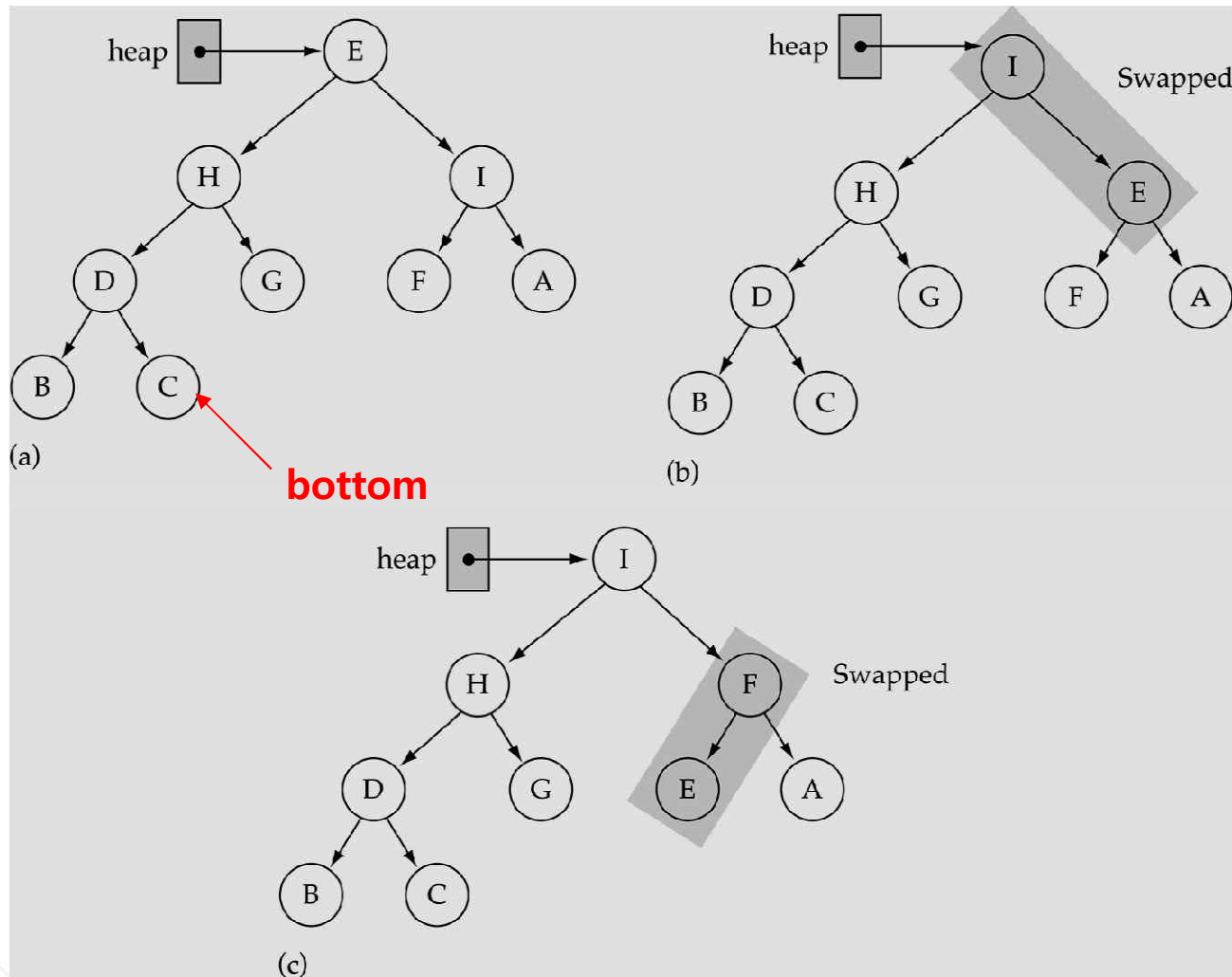
heap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

# Heap Specification

```
template<class ItemType>
struct HeapType {
    void ReheapDown(int, int);
    void ReheapUp(int, int);
    ItemType *elements; // dynamic array
    int numElements;
};
```

# The ReheapDown function



**Assumption:**

**heap property is  
violated at the  
root of the tree**

# ReheapDown function

```
template<class ItemType>
void HeapType<ItemType>::ReheapDown(int root, int bottom)
{
    int maxChild, rightChild, leftChild;

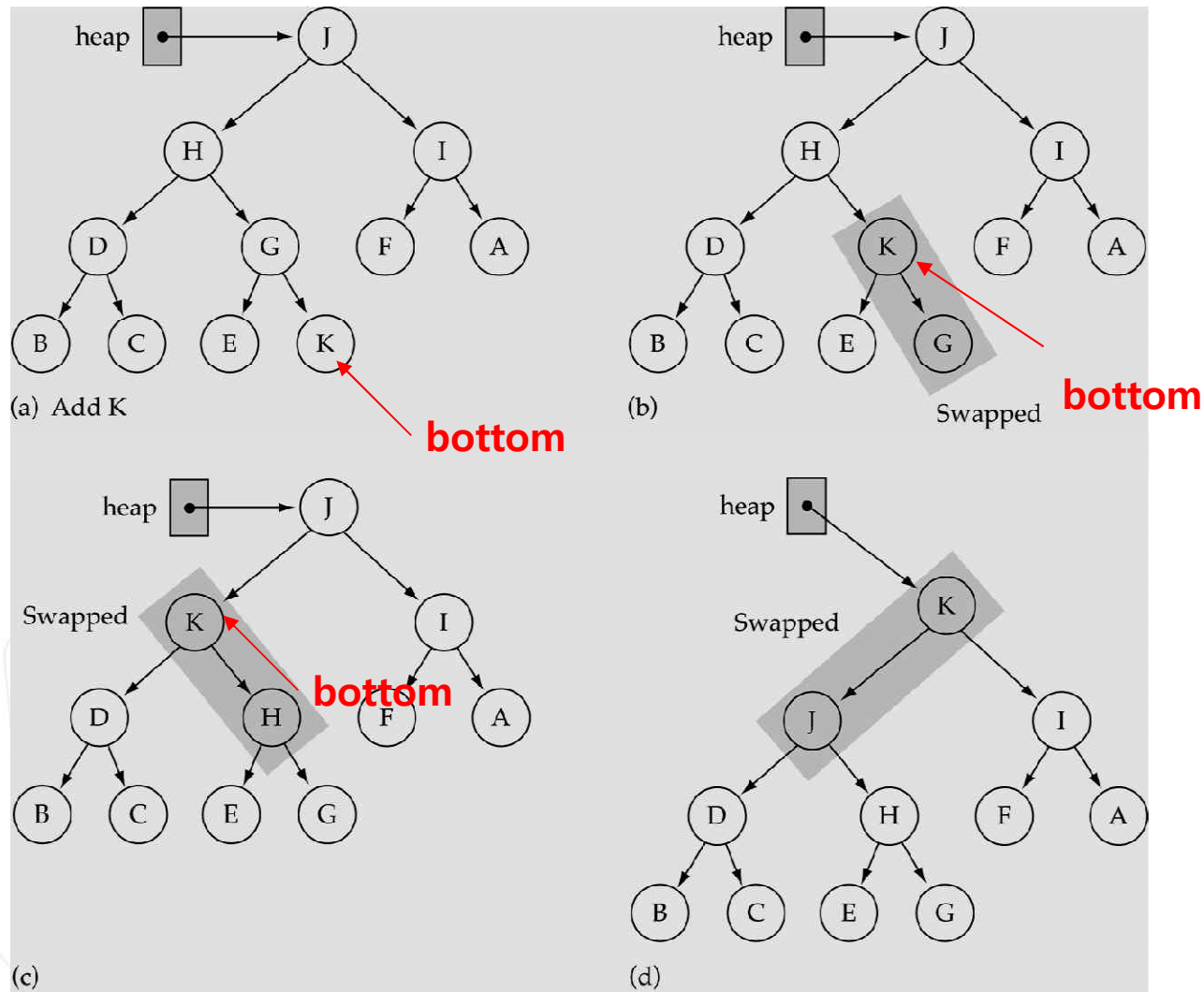
    leftChild = 2*root+1;
    rightChild = 2*root+2;

    if(leftChild <= bottom) { // left child is part of the heap
        if(leftChild == bottom) // only one child
            maxChild = leftChild;
        else { // two children
            if(elements[leftChild] <= elements[rightChild])
                maxChild = rightChild;
            else
                maxChild = leftChild;
        }
        if(elements[root] < elements[maxChild]) { // compare max child with
parent
            Swap(elements, root, maxChild);
            ReheapDown(maxChild, bottom);
        }
    }
}
```

rightmost node  
at the last level

13

# The ReheapUp function



**Assumption:**

**heap property is  
violated at the  
rightmost node  
of the last level  
of the tree**

# ReheapUp function

```
template<class ItemType>
void HeapType<ItemType>::ReheapUp(int root, int bottom)
{
    int parent;

    if(bottom > root) { // tree is not empty
        parent = (bottom-1)/2;
        if(elements[parent] < elements[bottom]) {
            Swap(elements, parent, bottom);
            ReheapUp(root, parent);
        }
    }
}
```

rightmost node  
at the last level

**$O(\log N)$**

# Priority Queues

- What is a priority queue?
  - It is a queue with each element being associated with a "priority"
  - From the elements in the queue, the one with the highest priority is dequeued first

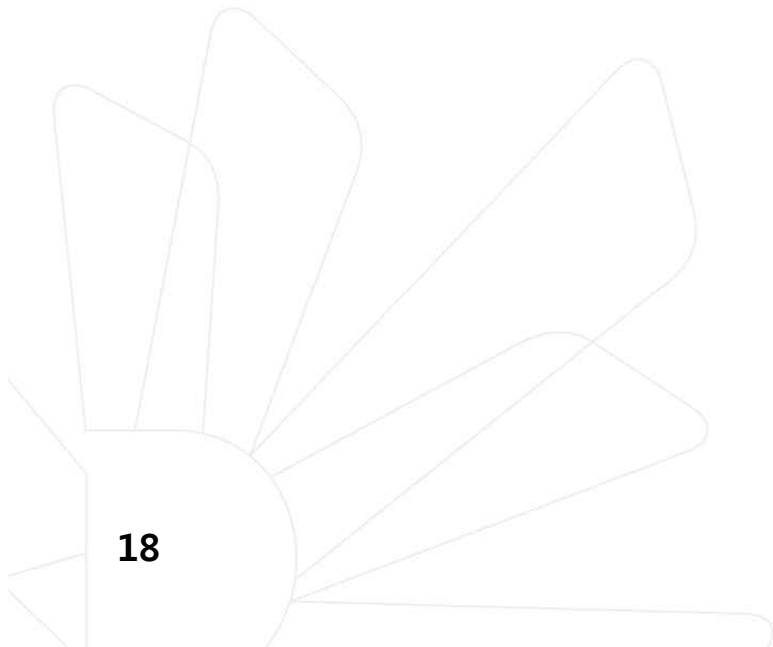


# Priority queue specification

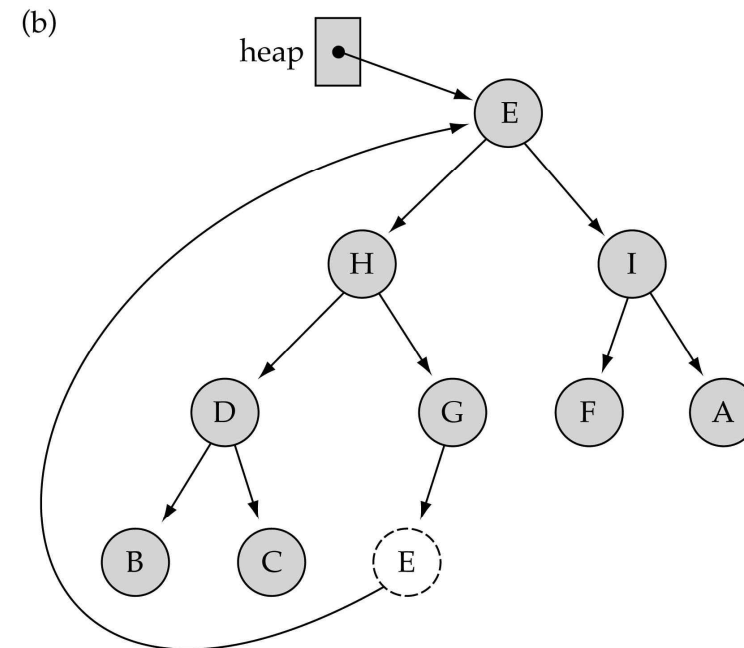
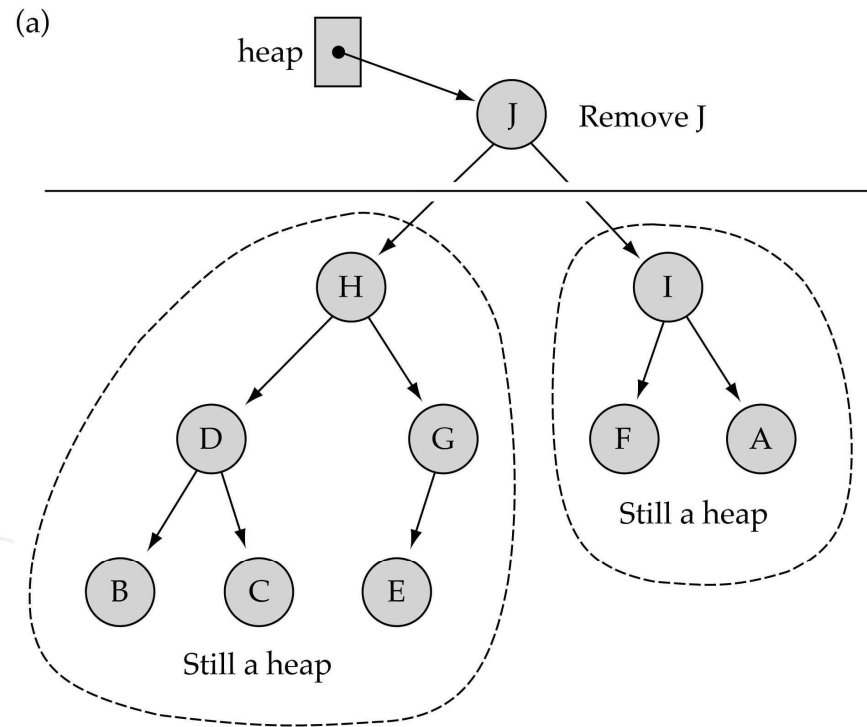
```
template<class ItemType>
class PQType {
public:
    PQType(int);
    ~PQType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Enqueue(ItemType);
    void Dequeue(ItemType&);
private:
    int numItems; // num of elements in the queue
    HeapType<ItemType> heap;
    int maxItems; // array size
};
```

# Dequeue: remove the largest element from the heap

- (1) Copy the bottom rightmost element to the root
- (2) Delete the bottom rightmost node
- (3) Fix the heap property by calling *ReheapDown*

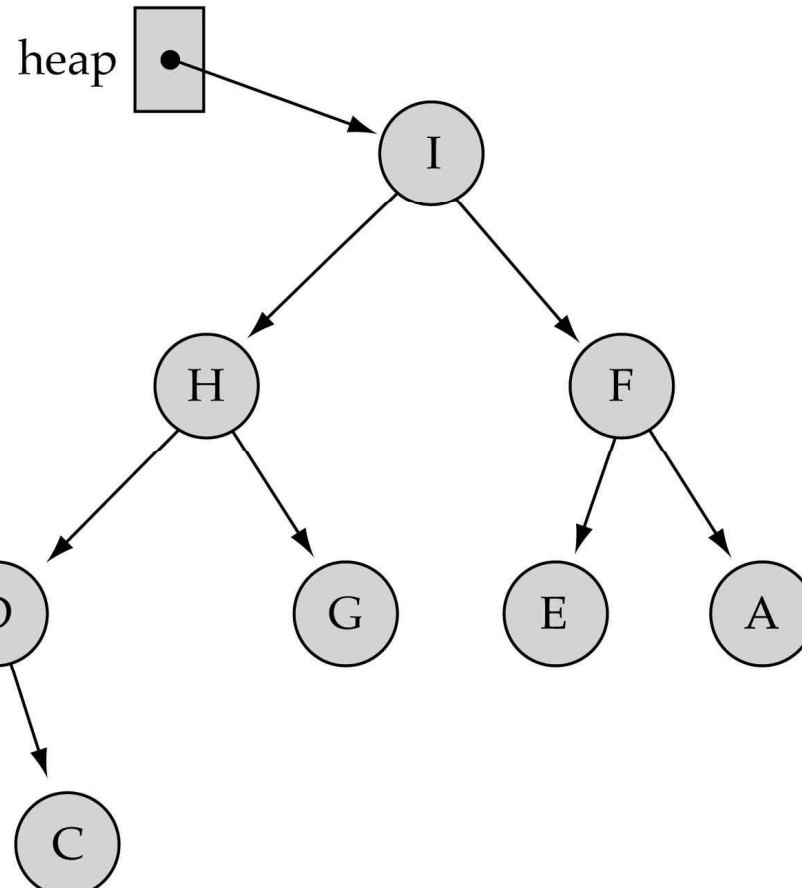


# Removing the largest element from the heap (cont.)



# Removing the largest element from the heap (cont.)

(c)



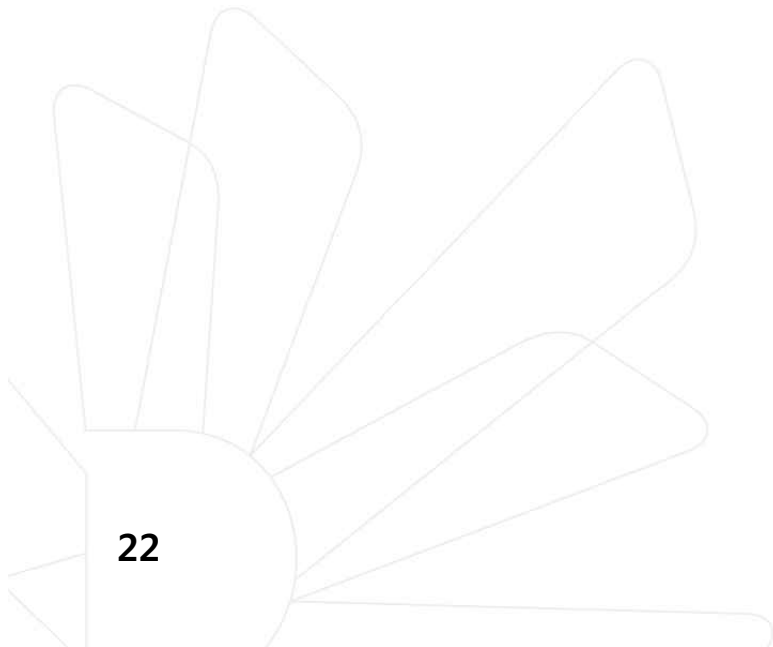
# Deque

```
template<class ItemType>
void PQType<ItemType>::Deque(ItemType& item)
{
    item = heap.elements[0];
    heap.elements[0] = heap.elements[numItems-1];
    numItems--;
    heap.ReheapDown(0, numItems-1);
}
```

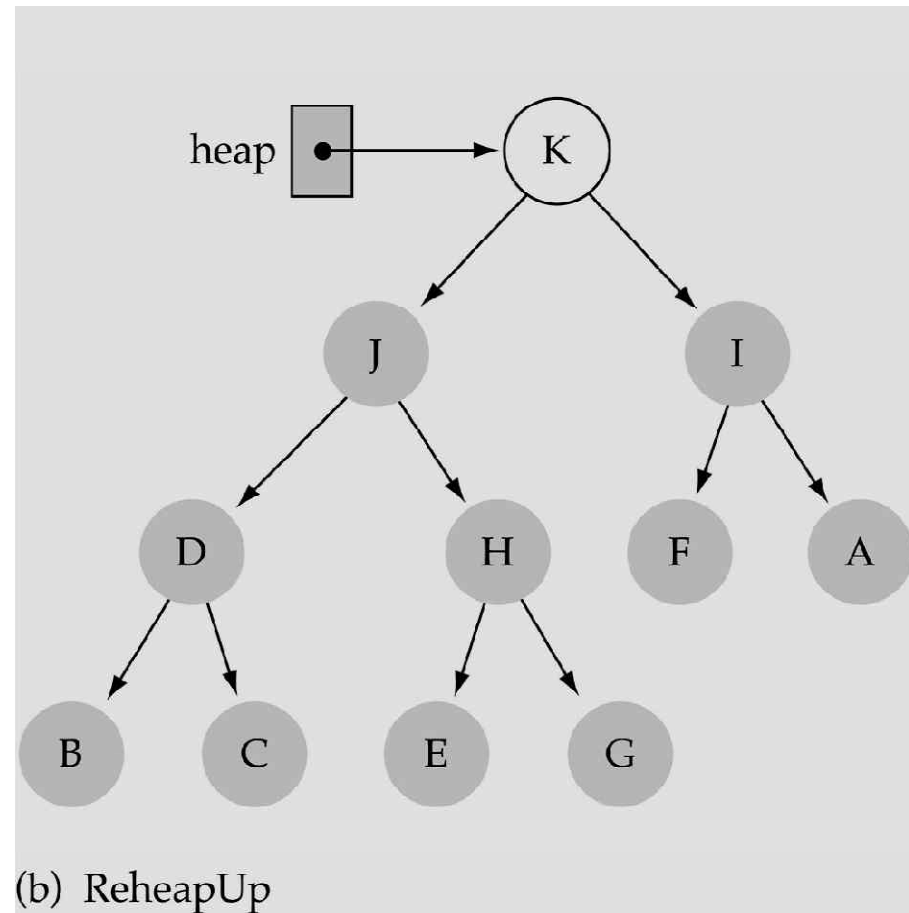
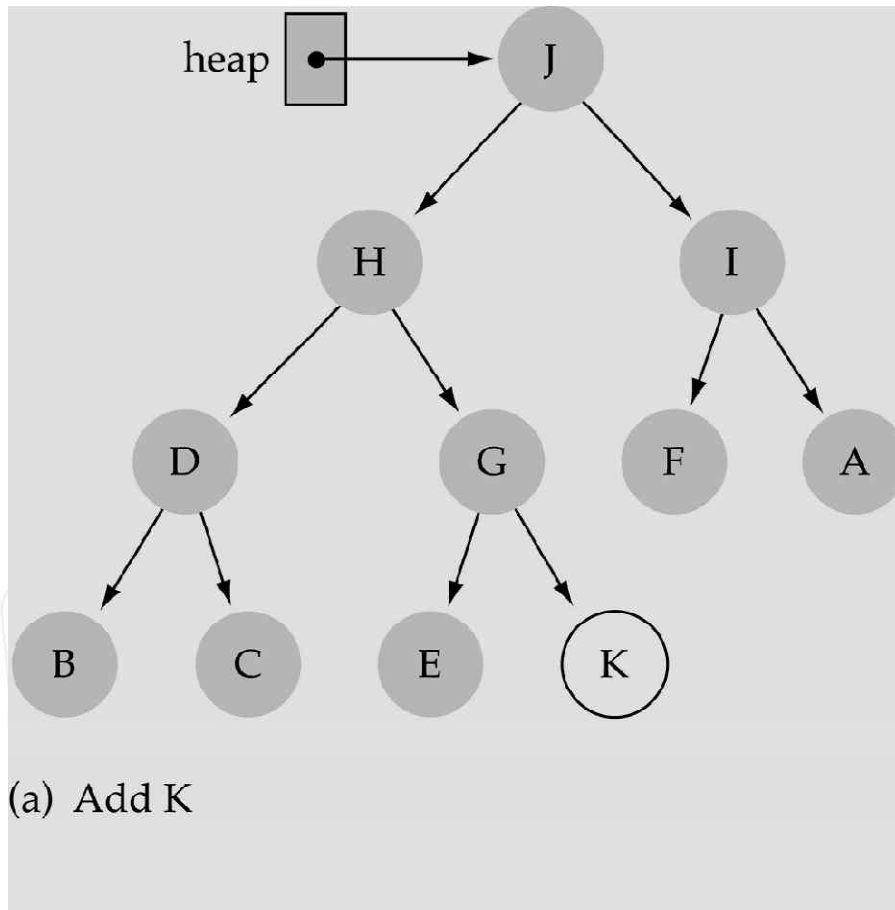
**bottom**

## Enqueue: insert a new element into the heap

- (1) Insert new element in the leftmost place at the bottom level (start new level if last level is full).
- (2) Fix the heap property by calling *ReheapUp* .

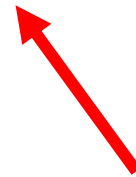


# Inserting a new element into the heap (cont.)



# Enqueue

```
template<class ItemType>
void PQType<ItemType>::Enqueue(ItemType newItem)
{
    numItems++;
    heap.elements[numItems-1] = newItem;
    heap.ReheapUp(0, numItems-1);
}
```



bottom



# Other Functions

```
template<class ItemType>
PQType<ItemType>::PQType(int max)
{
    maxItems = max;
    heap.elements = new ItemType[max];
    numItems = 0;
}
```

---

```
template<class ItemType>
PQType<ItemType>::MakeEmpty()
{
    numItems = 0;
}
```

---

```
template<class ItemType>
PQType<ItemType>::~~PQType()
{
    delete [ ] heap.elements;
}
```

## Other Functions (cont.)

```
template<class ItemType>
bool PQType<ItemType>::IsFull() const
{
    return numItems == maxItems;
}
```

---

```
template<class ItemType>
bool PQType<ItemType>::IsEmpty() const
{
    return numItems == 0;
}
```

# Comparing heaps with other implementations

- Priority queue using a sorted list



**$O(N)$  on the average!**

- Remove a key in  $O(1)$  time
- Insert a key in  $O(N)$  time

- Priority queue using heaps

- Remove a key in  $O(\log N)$  time
- Insert a key in  $O(\log N)$  time

**$O(\lg N)$  on the average!**